WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
(23 AND 24).USPT.	1
(L24 AND L23).USPT.	1

Database: US Patents Full-Text Database; US Pre-Grant Publication Full-Text Database; JPO Abstracts Database; EPO Abstracts Database; Derwent World Patents Index; IBM Technical Disclosure Bulletins

Search: L25

Refine Search

Recall Text | Clear

Search History

DATE: Tuesday, December 09, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L25	L24 and l23	1	L25
L24	flag$1	82967	L24
L23	L22 and l21	3	L23
L22	load$	897236	L22
L21	L20 and l19	3	L21
L20	JAVA	6661	L20
L19	l16 near4 class$2	10	L19
L18	reloadz$ near4 class$2	0	L18
L17	L16 and l8	1	L17
L16	reloading	8140	L16
L15	L14 and l1	1	L15
L14	L13 and l12	4	L14
L13	l4 near1 flag$1	1464	L13
L12	load$ near3 class$2	1569	L12
L11	load$ near 3 class$2	1	L11
L10	L9 and l6	1	L10
L9	6081665.pn.	1	L9
L8	L7 and l6	8	L8
L7	reset$ near4 flag	14187	L7
L6	L5 and l4 and l3 and l2 and l1	419	L6
L5	application or task$1	2035298	L5
L4	initializ$	103154	L4
L3	class and load$	82623	L3
L2	OOP or object oriented	10322	L2
L1	virtual machine$1	2647	L1

END OF SEARCH HISTORY

WEST

Generate Collection Print

L23: Entry 1 of 3 File: USPT Aug 7, 2001

DOCUMENT-IDENTIFIER: US 6272674 B1
TITLE: Method and apparatus for loading a Java application program

Abstract Text (1):
An apparatus and method for loading software into a Java virtual machine ("JVM") in a manner suited for real-time server applications. The software to be loaded is organized by Java package and class so that an application may be loaded in units of packages. Each package, and each class within a package, is loaded into the JVM in an order such that no package or class is loaded before the packages or classes upon which it depends. All software for an application is loaded into the JVM, and any compilation, optimization, or initialization takes place, prior to execution of the application program, so that no delays are incurred during such execution. Software loaded into the JVM, as well as attributes of that software, are identified. Versions of packages are compared when loading the packages to ensure compatibility. An "image" of loaded software is created, which image may be reused by the JVM in order to restart an application rapidly following a failure. A loader environment within the JVM contains information about all loaded applications, packages, and classes, their attributes, and their interrelationships.

Brief Summary Text (2):
The invention relates generally to Java application programs and, more particularly, to a method and apparatus for loading a Java application program to a Java virtual machine.

Brief Summary Text (6):
Modern telecommunication networks require complex, automated switching and, to that end, software programs are written to provide dependable performance and efficient use of resources, along with implementing service features and functions, such as Call Waiting, Caller ID, and the like. In such systems there may be different configurations depending on what types of transmission media are used, what types of users are served, and what mix of features are purchased. In order to perform dependably, all software required for operation must be loaded into the system and initialized before the system begins its normal processing; otherwise, unpredictable variations in performance, and even unacceptable delays, might be experienced as needed software is identified, loaded, and initialized before processing can continue.

Brief Summary Text (7):
A computer language for implementing software for such systems is "Java." Java was introduced by Sun Microsystems, Inc., of Palo Alto, Calif., and has been described as an object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, and dynamic computer language. A key feature of Java with respect to this invention is its ability to load software dynamically. In many programming systems today, entire software applications are constructed (i.e., software modules are linked together) as a unit. Java, however, allows software modules to be loaded and linked into a running program environment, known as a Java virtual machine (JVM). Thus, changing one module need not involve re-linking the entire application. Furthermore, applications may be extended by adding modules to the application without interrupting execution of the application. This capability makes Java very useful in the construction of server software applications.

Brief Summary Text (8):

In the Java programming language, individual source files describing classes are compiled to produce class files, which are the most basic unit of software introduced into a system. As used herein, the term "class" refers to a generalized category that describes a group of more specific methods that can exist within it, and are comparable in concept to the types of "pigeonholes" used to organize information. The term "method" as used herein denotes a procedure or a function. Data and methods, taken together, generally serve to define the contents and capabilities of an object.

Brief Summary Text (9):
Classes may be grouped into "packages," but packages are not presently a unit by which software code is loaded into a system. In a standard Java virtual machine (JVM), classes are typically loaded one at a time from class files, or perhaps from a compressed archive containing a number of class files within it, possibly from unrelated packages. In accordance with a method of loading often referred to as "lazy loading," a class is not loaded until that class is needed by the JVM. Any necessary initialization for that class is similarly deferred for as long as possible. These techniques are suitable for software systems, such as "applets" in web browsers, that are primarily user-interactive. If all software that might possibly be needed were to be loaded and initialized before the applet could interact with the user, the user would experience an unacceptable delay.

Brief Summary Text (10):
While lazy loading is appropriate for non-real time systems, such as that described above, lazy loading of software applications into the JVM is usually not appropriate for real-time server applications. The unpredictable performance and unexpected latency associated with lazy loading is often intensified because Java classes are commonly dependent on other classes. In many cases, in order to load one class, if other classes upon which the one class depends have not yet been loaded, the JVM will stop loading the one class while it attempts to load the other classes.

Brief Summary Text (11):
In accordance with conventional JVM technology, application software is executed by first loading a "key" class and then executing a particular method of that class. In a stand-alone application, the key class is a class with a "main" method which provides a starting point for the program. In an applet in a browser window, the key class is derived from the base applet class and is loaded, and a "start" method is called. In either the stand-alone application or the applet, once the key class is loaded, the remaining classes are identified and loaded as required. In some cases, security controls are used to constrain class loading. For example, an applet can only load classes from the same server from which the applet itself was initially loaded. The JVM does keep track of classes loaded, but does not keep track of packages loaded, nor does it keep track of certain attributes of classes and packages that might be of interest.

Brief Summary Text (12):
For reasons of manageability, in a large-scale system having at least a single JVM, entire applications may be loaded more efficiently as collections of packages, each of which packages encapsulates a collection of classes. This reduces, by an order of magnitude, the number of software objects that must be managed. Furthermore, if more than one application is loaded into a single JVM of the system, some packages may be shared between the applications and so need only be loaded once, reducing load times and saving memory space. In such a case, the package becomes the unit of software loaded into the system, rather than the individual class file. It thus becomes more important to ensure that the package, as a concrete unit of software, can be immediately loaded from a package load file that contains all classes belonging to the package.

Brief Summary Text (13):
Another feature of large-scale systems is that some software objects that make up the configuration of a running system may not have been developed, tested, and packaged at the same time. Instead, the objects may be of different vintages, and include some components that have remained unchanged for a long time, and some other components that continually change as the software is further developed and improved. As a result, it is often important to know what software objects are

loaded into such a system, and to be able to ensure that only objects of compatible vintages are combined together.

Brief Summary Text (14):
Accordingly, a continuing search has been directed to the development of methods for loading classes without incurring unpredictable performance and unexpected latency associated with lazy loading, for loading packages only as needed to avoid increased load times and depleting memory unnecessarily, and for ensuring that software objects loaded in a system are of compatible vintages.

Brief Summary Text (16):
According to the present invention, Java software applications are loaded into a Java virtual machine (JVM) in a manner suited for real-time server applications. The software to be loaded is organized by Java package and class so that an application may be loaded in units of packages. Each package, and each class within a package, is loaded into the JVM in an order such that no package or class is loaded before the packages or classes upon which it depends. All software for an application is loaded into the JVM, and any compilation, optimization, or initialization takes place, prior to execution of the application program, so that no delays are incurred during such execution. Software loaded into the JVM, as well as attributes of that software, are identified. Versions of packages are compared when loading the packages to ensure compatibility. An "image" of loaded software is created, which image may be reused by the JVM in order to restart an application rapidly following a failure. A loader environment within the JVM contains information about all loaded applications, packages, and classes, their attributes, and their interrelationships.

Drawing Description Text (3):
FIG. 1 is a block diagram illustrating a Java Virtual Machine ("JVM");

Drawing Description Text (4):
FIG. 2 is a block diagram illustrating an application program to be loaded onto the JVM of FIG. 1;

Drawing Description Text (5):
FIG. 3 is a block diagram illustrating a loader environment within the JVM of FIG. 1; and

Detailed Description Text (2):
Referring to FIG. 1 of the drawings, the reference numeral 100 generally designates a Java Virtual Machine ("JVM") embodying features of the present invention. The JVM 100 may be implemented on any of a number of different computer platforms (not shown), such as a personal computer ("PC"), a Macintosh computer, a Unix workstation, or the like, running any of a number of different operating systems, such as Unix, Windows, MacOS, or the like. Such computer platforms and operating systems are considered to be well-known and will, therefore, not be described in further detail.

Detailed Description Text (3):
The JVM 100 includes, within an electronic memory (not shown) of the computer, a main memory 102 with a heap 104, and a JVM internal memory 106. The main memory 102 is an environment within which a Java application program 120, described further below, may be executed. The internal memory 106 is partitioned to include a logical area of memory, designated as a loader environment 200, for loading the application program 120. The internal memory 106 is used to operate the JVM 100 and is not generally accessible to a Java program running in that JVM for safety and security reasons. The JVM 100 also includes a function component 110 for providing a garbage collection function 110a, a system interface 110b, an execution engine 110c (for executing instructions contained in methods of loaded classes), and the like, including threads (not shown) as defined by the architecture of the JVM 100.

Detailed Description Text (4):
When the JVM 100 runs the Java application program 120, the memories 102 and 106 are used to store Java components, such as bytecodes (i.e., method bodies) and other information extracted from a loaded class file (described below), objects the

program instantiates, parameters to Java methods, return values, local variables, intermediate results of computations, and the like. When a class instance or array is created in a running Java application program 120, the memory for the new class is allocated from the heap 104 portion of the main memory 102.

Detailed Description Text (6):
As discussed further below, FIG. 2 exemplifies the application program 120 as comprising data structures for an application control file 122, and three package files 124. The three package files 124 are substantially similar to each other in a structural sense and, for the sake of conciseness, will therefore be described below representatively as the package file 124. Each package file 124 is depicted as comprising data structures for three Java class files 128 which are substantially similar to each other in a structural sense and, for the sake of conciseness, will therefore described representatively as the class file 128. The package file 124 may also contain a manifest 140. As indicated by the ellipses, the application program 120 may comprise more or less than three package files 124, and more or less than three class files 128 within each package file 124. It should be noted though that the application program 120 is not a file, as such, containing within it the application control file 122, package files 124, though the application control file 122 does contain within it the identity of the package files included within the application program 120. The package files 124 do contain within them the class files 128.

Detailed Description Text (7):
Each class file 128 contains everything the JVM 100 needs to know about one Java class or interface. This information is set out in a well-defined class file format to ensure that any Java class file can be loaded and correctly interpreted by any JVM 100, no matter what computer system produced the class file 128 or what system hosts the JVM 100. The class file 128 includes a "magic number" (not shown), such as 0xCAFEBABE, which identifies it as a Java file. Each class file also includes a version number (not shown), a constant pool 130, a method_info portion 132, and an attributes portion 134, described below. Class files are considered to be well-known in the art and are described, for example, in a JVM specification entitled "The Java Virtual Machine" by Tim Lindholm and Frank Yellin (1997), ISBN 0-201-63452-X, which is commercially available from Sun Microsystems, Inc. or at the web address http://www.aw.com/cp/javaseries.

Detailed Description Text (11):
To build the application program 120, the source files to all classes must be compiled. The source files may be compiled using an existing software development tool, such as the Java Development Kit (JDK), which is commercially available from Sun Microsystems, Inc. A "key" class of the application program 120 must then be identified, and some method of the key class must be executed (possibly on a new instance object) to begin executing the application program.

Detailed Description Text (13):
From the list of classes, a list of packages may be obtained, wherein each class is a member of one and only one package. Each package identified will in turn have a list of its constituent classes, which list is generated by from the class files, or might be obtained from a database in an advanced software development environment or library system where the source files are maintained. The list of needed packages may be computed in a manner similar to that used for the classes, as described above. This list of packages is generated in a certain order such that each package loads before packages that depend on it, and is stored in the application control file 122. The order is determined as a by-product of the computation of the list of needed packages by the aforementioned order determination algorithm. The application control file 122 may also identify the "key" class and possibly other attributes of the application program 120, such as the date of its construction, security information, and the like.

Detailed Description Text (14):
For each package required, a package file 124 is generated containing the class files 128 of the package file 124, in such an order that each class file precedes classes that may depend on it. The package file 124 may also contain a manifest 140 providing security information for the classes and the like. The package load file

may be in the format of a Java archive (JAR) file (not shown), or some other format. The JAR file format is well-known and is described in greater detail, for example, in a document entitled "jar-The Java Archive Tool" which is available at the web address http://www.javasoft.com/. This order may be determined as a by-product of computing the class needs, as described above with respect to the order determination algorithm, or may be computed anew through the same or a similar algorithm, applied only to the classes which constitute the package. It should be noted, however, that packages which may be part of a "standard library" associated with the JVM 100 need not have package files created for them; it is assumed that such packages are resident with the JVM and do not require loading to the JVM through this method.

Detailed Description Text (15):
As mentioned above with respect to FIG. 1, the JVM internal memory 106 includes a logical area of memory, designated as a loader environment 200, for loading the application program 120. The application program 120 includes at least one package, depicted in FIG. 2 as the package files 124, each of which have at least one type, i.e., at least one class file 128 and corresponding interface (not shown) having fully qualified names. The loader environment 200 catalogs each application, package, and class loaded, along with their relationships and other attributes. The relationships define, for example, which objects (e.g., application program 120, package files 124, class files 128, and the like) contain or are contained by which other objects, which objects require or are required by which other objects, and the like. Attributes for packages and classes include author, compile date, package build date, version of the package or class, version(s) of required packages or classes that are known to be compatible (or incompatible), and the like.

Detailed Description Text (16):
Referring back to FIG. 1, the loader environment 200 is configured for storing metadata describing attributes of the application programs 120 (FIG. 2), such as the version number, compile date, and the like, and attributes of the classes and packages loaded as part of those application programs. When the JVM 100 loads a Java application program 120, the JVM 100 parses attribute information from the application control file 122 (FIG. 2). Such attributes for the packages may be stored within the package file 124 within the manifest 140 (FIG. 2), (e.g., in the JAR file format described in the aforementioned JAR specification). For each class, attribute information is contained in the class file 128 (FIG. 2) as described above. Effectively, the JVM 100 builds within the loader environment 200 a collection of information about all loaded software, and makes such information available to programs running on the JVM via an application programming interface (API) in a manner well-known in the art.

Detailed Description Text (17):
Such a collection of information in the loader environment 200 is exemplified in FIG. 3 as comprising a list 304 of applications 304a, 304b, and 304c, a list 306 of packages 306a, 306b, 306c, and 306d, and a list 308 of classes 308a, 308b, 308c, and 308d effective as data structures for cataloging the installed software of the loader environment 200. The number of applications, packages, and classes making up the loader environment 200 may vary from the number shown in FIG. 3. Relationships are also cataloged, as indicated by the arrows 310, such as between the application A.sub.1 304a and the package P.sub.1 306a. The loader environment 200 also contains a JVM software environment 302 (FIG. 4), which is part of the JVM 100. The software environment 302 contains such data as the methods of the classes, their types and arguments, and the like, as well as by-products such as native code generated by a just-in-time (JIT) compiler, and the like, stored in a manner well-known to the art. The class elements 308a-308d in the loader environment 200 may refer back to the JVM software environment 302.

Detailed Description Text (18):
The size of the loader environment 200 need not be fixed. As the Java application program 120 runs, the JVM 100 can expand and contract the loader environment 200 to fit the needs of the application. Generally, users or programmers may specify an initial size for the loader environment 200, as well as a maximum or minimum size.

Detailed Description Text (19):

FIG. 4 is a flow chart of steps implemented in the operation of loading the Java software packages 124 of an application program 120 in accordance with the present invention. Accordingly, in step 400, given the application 120 to be loaded to the JVM 100, a list of packages that are required for the application is derived. The package list may be derived using any available technique, such as, for example, by using a configuration management system (not shown).

Detailed Description Text (20):
In step 402, a class load order is determined first within each package for all of the class files 128 within that respective package, so that when each class is loaded, any classes on which that respective class depends will have been previously loaded. Similarly, a package load order is determined for each of the package files 124, as described above, so that when each package is loaded, any packages on which that respective package depends will have been previously loaded.

Detailed Description Text (21):
In step 404, any metadata associated with a class is incorporated into its respective class file 128. The class file 128 and any metadata associated with a package, including any security information, are then incorporated into a respective package file 124. These package files 124 may then be placed in some repository, such as a disk directory, web site, database, or the like, from which they may be loaded when required.

Detailed Description Text (22):
In step 406, the application control file 122 for the entire application program 120 is generated. The application control file 122 includes a list of the package files 124 in the order that they are to be loaded into the application program 120, and may also include other information, such as a "key" class, security information, and the like. The application control file 122 is also stored so that it may be used to load the application program 120; however, it need not be stored together with the package files.

Detailed Description Text (23):
In step 408, operation of the Java virtual machine 100 is initiated. The application program 120 to be loaded may be passed as a parameter to the JVM 100 by some means dependent on the operating system, or the JVM may wait for a command to load that is supplied externally, e.g., through a network interface.

Detailed Description Text (25):
In step 412, the JVM commences to load and process the application 120 beginning with the first package file 124 on the list extracted in step 410, i.e., the package file 124 that does not depend on any other listed package file. Prior to actually loading the package, the JVM 100 first looks up the respective package in the loader environment 200 to determine whether the package has been previously loaded (such as, for example, in a standard library which may have been previously loaded). If the current package has not been loaded, then the JVM 100 verifies that any other packages on which the current package depends are loaded and are compatible. If such other packages are not loaded or are incompatible, then the JVM 100 stops loading and gives an error message, e.g., by displaying a message on a terminal screen (not shown), printing the message on a printer (not shown), or the like. Otherwise, if such other packages are loaded and are compatible with the package being loaded, then the JVM 100 opens the package file 124, by reading a disk file (not shown), by opening a network connection to download the package file 124 from another machine (not shown), by acquiring it from a database (not shown), or the like. The JVM 100 updates its loader environment 200 with data from the package file 124, which data represents attributes of the package, and of the relationship of the package being loaded to other packages.

Detailed Description Text (26):
In step 414, the JVM loads the first class file 128 from the package file 124 being loaded, decompresses it as necessary, verifies any security information, and links it into the JVM loader environment 200. This method by which class files 128 are loaded into the JVM, such as the JVM 100, is considered to be well-known in the art and will therefore not be described further. Additionally, attributes of the class, as well as its relationship to the package in which it is contained, are entered

into the loader environment 200.

Detailed Description Text (27):
In step 416, if there are additional class files 128 in the package file 124 to load, execution proceeds to step 418; otherwise, execution proceeds to step 420. At step 418, the next class file 128 in the package file 124 being loaded is loaded in the manner described with respect to step 414, and the loader environment 200 is updated. Upon loading the next class file 128, execution returns to step 416.

Detailed Description Text (28):
In step 420, any additional processing, such as pre-compilation to native code, optimization, execution of the initialization routines for the classes, or the like, required for the classes 128 loaded from the package file 124 is performed in a manner well-known in the art.

Detailed Description Text (29):
In step 422, a determination is made whether there are additional package files 124 to load in the application program 120. If it is determined that there are additional package files 124 to load in the application program 120, then execution proceeds to step 424 wherein the next package in the list of packages extracted in step 410 is loaded. Following step 424, execution returns to step 414. If, in step 422, it is determined that there are no additional package files 124 to load in the application program 120, then execution proceeds to step 426, wherein execution of the application program 120 on the JVM 100 commences in a manner well-known in the art.

Detailed Description Text (30):
By the use of the present invention, a Java software application program may be efficiently preloaded onto the JVM 100 to thereby eliminate "lazy loading" and enhance the performance of real-time systems. The present invention also provides a basis for determining what software is loaded onto a running, as the loader environment 200 contains a list of the running application(s), packages, classes, their attributes, and their interrelationships. A JVM 100 using the method of this invention may provide an application programming interface (API) or some other method by which a program running on such a JVM 100 may query the loader environment 200 and inspect the information stored therein, or by which an external program may query the JVM for that information, or both.

Detailed Description Text (31):
It is understood that the present invention can take many forms and embodiments. Accordingly, several variations may be made in the foregoing without departing from the spirit or the scope of the invention; for example, more than one application may be loaded into a single JVM 100. In another example, an image of the loaded application may be stored in a non-volatile medium. That is, the state of the loader environment 200, including all classes, packages, and information about the classes and packages, may be written out to a non-volatile medium, such as a hard disk file. This information would include the code (including compiled or optimized code) for methods of the classes so written out. Then, in order to restart the JVM 100 with that same software at a later time, the disk file may be simply read in, allowing the JVM to bypass the steps 400-418 and 422-424 in the above description of the flow chart shown in FIG. 4. Instead, the JVM 100 would need only to reinitialize each class (thus recreating any initial data in the heap 104) and commence program execution (steps 420 and 426, respectively). Such a technique would greatly enhance the speed with which a JVM 100 could be restarted after a failure, such as a hardware crash. In still another example, the JVM 100 may provide an interface, such as a network interface, an inter-process communication interface configured for a particular operating system, or the like, through which interface an external program may inspect the data structures of the application program, packages, and classes loaded in the JVM.

CLAIMS:

1. A method for loading a Java application program onto a Java Virtual Machine ("JVM"), comprising the steps of:

(a) identifying package files required for operation of the application program, each of which package files comprises at least one class file;

(b) determining a package load order in which each respective package file may be loaded before any package file is loaded that depends on the respective package file;

(c) determining a class load order in which each respective class file within each package file may be loaded before any class file is loaded that depends on the respective class file; and

(d) loading into the JVM the package files in the package load order, and the class files within each package file in the class load order.

2. The method of claim 1 wherein the step of determining a package load order further comprises recursively calculating the transitive closure of dependencies of the package files.

3. The method of claim 1 wherein the step of determining a class load order further comprises recursively calculating the transitive closure of dependencies of the class files.

4. The method of claim 1 wherein the application program contains a key class, and the step of determining a package load order further comprises identifying all class files required by the key class, and recursively identifying the requirements of each of the class files, until no new class files are known to be required.

5. The method of claim 1 wherein the application program contains a key class, and the step of determining a class load order further comprises identifying all class files required by the key class, and recursively identifying the requirements of each of the class files, until no new class files are known to be required.

6. The method of claim 1 wherein the step of loading further comprises, for each respective package file, the steps of determining whether the respective package file has previously been loaded; and upon a determination that the respective package file has not been loaded, loading the respective package file.

7. The method of claim 1, wherein the step of loading further comprises the step of compiling all loaded class files.

8. The method of claim 1, wherein the step of loading further comprises the step of optimizing all loaded class files.

9. The method of claim 1, wherein the step of loading further comprises the steps of initializing all loaded class files.

10. The method of claim 1, wherein the step of loading includes comparing versions of loaded package files with requirements of the package files to be loaded, to determine version compatibility between package files.

11. The method of claim 1 wherein the JVM includes a loader environment, and the step of loading package files into the JVM further comprises loading each package file into the loader environment of the JVM.

12. The method of claim 1 wherein the JVM includes a loader environment, and step of loading of package files into the JVM further comprises loading each package file into the loader environment of the JVM; and storing in non-volatile memory or other media the contents of a loader environment, such that the contents may be retrieved by a fresh invocation of the JVM in order to execute the program without individually reloading each application, package file, and class file.

13. The method of claim 1 further comprising the step of generating with respect to each package file a package load file identifying in the class load order the class files contained within the respective package file.

14. The method of claim 1 further comprising the steps of generating an application control file identifying the package files in the package load order, and reading the application control file into the JVM; and the step of loading further comprises loading each package file into the JVM in the package load order stored in the application control file, wherein the loading of each respective package file further comprises loading each class file of the respective package file into the JVM in the class load order.

15. A method for loading a Java application program to a Java Virtual Machine ("JVM") residing on a computer, the Java application program including a plurality of package files, each of which package files includes a plurality of class files, the method comprising:

determining a class load order in which each respective class file within each package file may be loaded before any class file is loaded that depends on the respective class file;

determining a package load order in which each respective package file may be loaded before any package file is loaded that depends on the respective package file;

storing the package load order in an application control file attached to the application program;

reading the application control file into the JVM; and

loading each package file into the JVM in the package load order stored in the application control file, wherein the loading of each respective package file further comprises loading each class file of the respective package file into the JVM in the class load order.

16. The method of claim 15 wherein the step loading each respective package file further comprises compiling the respective package file.

17. The method of claim 15 wherein the step loading each respective package file further comprises initializing the respective package file.

18. The method of claim 15 wherein the step of loading each respective package file further comprises optimizing the respective package file.

19. The method of claim 15 wherein the step of determining a class load order further comprises recursively calculating the transitive closure of the dependencies of the application program.

20. The method of claim 15 wherein the step of determining a package load order further comprises recursively calculating the transitive closure of the dependencies of the application program.

21. The method of claim 15 wherein the JVM includes a loader environment, and step of loading each package file into the JVM further comprises loading each package file into the loader environment of the JVM.

22. The method of claim 15 wherein the JVM includes a loader environment containing information about software objects loaded on the computer through the JVM, and step of loading each package file into the JVM further comprises loading each package file into the loader environment of the JVM.

23. A Java Virtual Machine ("JVM") operable on a computer and having a loader environment containing information about software objects which may be loaded onto the computer through the JVM, the loader environment comprising:

an application data structure defined on an electronic memory of the computer, the application data structure being configured for receiving at least one application program;

at least one package data structure defined on the electronic memory of the computer

for identifying each respective package of the application data structure in such an order that, when the packages are loaded, no respective package is loaded before the packages upon which the respective package depends are loaded; and

at least one class data structure defined on the electronic memory of the computer for identifying each Java respective class data structure of the package data structure in such an order that, when the class data structures are loaded, no respective class data structures is loaded before the class data structures upon which the respective class data structure depends are loaded.

25. The JVM of claim 23 wherein the application data structure further comprises attributes belonging to applications loaded onto the JVM, the package data structure further comprises attributes belonging to packages loaded onto the JVM package data structure, and the class data structure further comprises attributes belonging to class data structures loaded onto the JVM.

WEST

Help | Logout | Interrupt

Main Menu | Search Form | Posting Counts | Show S Numbers | Edit S Numbers | Preferences | Cases

Search Results -

Term	Documents
(20 AND 14).USPT.	8
(L20 AND L14).USPT.	8

Database: US Patents Full-Text Database
US Pre-Grant Publication Full-Text Database
JPO Abstracts Database
EPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Search: L21

Refine Search

Recall Text | Clear

Search History

DATE: Wednesday, December 10, 2003 Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=USPT; PLUR=YES; OP=ADJ			
L21	L20 and l14	8	L21
L20	reset$ near4 flag$	14234	L20
L19	reset near4 flag	13603	L19
L18	L17 and l14	6	L18
L17	reset near4 flag$	13654	L17
L16	L15 and l14	2	L16
L15	reset near5 flag and reinitializ$	674	L15
L14	L12 and l11 and l10 and l8 and l7	419	L14
L13	L12 and l11 andl10 and l9 and l8 and l7	0	L13
L12	application$1 or task$1	2035492	L12
L11	initializ$	103154	L11
L10	class and load$	82623	L10
L9	class or load$	1068872	L9
L8	OOP or object oriented	10322	L8
L7	virtual machine$1	2647	L7
L6	L5 and l2 and l3	1	L6
L5	6523168.pn.	1	L5
L4	L3 and l2 and l1	1	L4
L3	flag$1	82967	L3
L2	initializ$	103154	L2
L1	6081665.pn.	1	L1

END OF SEARCH HISTORY

WEST

Generate Collection Print

L21: Entry 1 of 8 File: USPT Nov 25, 2003

DOCUMENT-IDENTIFIER: US 6654948 B1
TITLE: Methods and apparatus for partial and consistent monitoring of object-oriented programs and systems

Abstract Text (1):
A technique for monitoring events generated by an object-oriented system comprises the steps/operations of: (i) monitoring events which describe executed operations associated with the object-oriented system; and (ii) applying one or more sequencing rules when reporting a subset of the monitored events, the one or more sequencing rules substantially ensuring consistent reporting of the subset of monitored events. Preferably, monitoring continues when event reporting is at least partially disabled. Further, the monitoring step/operation may include dividing the monitored events into categories. One category may include entity events, an entity event defining an existence status of a given event. Another category may include activity events, an activity event defining an operation associated with a given event. Still further, the entity events and activity events may be further divided into at least one of an object event category, an execution event category, a type event category and a synchronization event category. The sequencing rules are applied to maintain substantial consistency with respect to information associated with the categories.

Brief Summary Text (2):
The present invention generally relates to computer programming and, in particular, to monitoring object-oriented programs.

Brief Summary Text (4):
Monitoring is that activity where a program execution environment reports to external listener subsystems what is happening during the program execution. Monitoring enables many different tools for performing! different tasks. Examples of such tools are profiling tools for performance enhancements, tracing tools for program understanding, or debugger tools for debugging. Monitoring is generally costly because: (i) amount of information produced is fairly large; (ii) the overhead of monitoring is high; or (iii) a combination of both. Consequently, many systems support partial monitoring where only a subset of the information is produced. There are many different ways for expressing which subset is of interest. However, no matter what conventional method is used to partially monitor a system, the resulting partial monitoring can result in the generation of an inconsistent subset of the complete information. Such an inconsistent subset of information can lead to incorrect interpretation of the program execution behavior by the various information processing tools and lead to erroneous conclusions.

Brief Summary Text (8):
In one aspect of the invention, a method of monitoring events generated by an object-oriented system comprises the steps of: (i) monitoring events which describe executed operations associated with the object-oriented system; and (ii) applying one or more sequencing rules when reporting a subset of the monitored events, the one or more sequencing rules substantially ensuring consistent reporting of the subset of monitored events. Preferably, monitoring continues when event reporting is at least partially disabled. Further, the monitoring step may include dividing the monitored events into categories. One category may include entity events, an entity event defining an existence status (e.g., object creation, object reclamation, etc.) of a given event. Another category may include activity events, an activity event defining an operation associated with a given event. Still further, the entity events and activity events may be further divided into at least one of an object

event category, an execution event category, a type event category and a synchronization event category. The sequencing rules are applied to maintain substantial consistency with respect to information associated with the categories.

Brief Summary Text (10):
Advantageously, the methodologies of the invention may produce substantially consistent information in the context of partial reporting (i.e., reporting a subset of the events being monitored) with respect to a program being executed by the object-oriented system. Selection of the subset of information to be reported may be made using either dynamic or static filtering criteria. The information may be used by tools such as, for example, program analyzers or program visualizers to correctly interpret the program execution and solve performance, program understanding and correctness problems.

Brief Summary Text (11):
It is to be appreciated that the term "partial monitoring" as used in accordance with the invention may be thought of as referring to the perspective of the external listener (e.g., tool). That is, while a subset of the monitored events are reported to the listener in accordance with the one or more sequencing rules, the methodology of the present invention preferably monitors substantially all events associated with the object-oriented system. However, due to the subset reporting, it appears to the external listener that the listener is partially monitoring the object-oriented system. As mentioned, one advantage over the prior art is that the sequencing rules substantially ensure consistent reporting of the subset of monitored events.

Drawing Description Text (4):
FIG. 2 is a visual representation illustrating object-oriented program execution through time;

Detailed Description Text (2):
As mentioned, monitoring is that activity where a running system reports to external listeners what is happening inside the system. A system usually reports its activity through events. Depending on the object-oriented system that is considered, these events can be different in nature. For example, a program execution environment can report events related to the execution of the program. The events reported can be read accesses to data structures, write accesses to data structures, the beginning of execution of a method, the end of execution of a method, etc. A high-level block diagram illustrating the context of the invention is shown in FIG. 1. As shown, a program 2 is executed within a program execution environment 4. The environment 4 includes a program execution subsystem 6 and an integrated monitoring subsystem (agent) 8. The program execution subsystem 6 feeds the monitoring agent 8 different types of events and event information regarding program execution. The monitoring agent portion of the program execution environment reports events in a form that is readable and understandable by external listener subsystems such as, for example, program event processing tools 10. The event reporting may be controlled by an event reporting controller 12. That is, the controller 12 directs the monitoring agent 8 what to report and whether to report (i.e., reporting on) or not report (i.e., reporting off) information to the processing tool 10. It is to be appreciated that this invention deals with particular methodologies of developing and providing such an integrated monitoring subsystem.

Detailed Description Text (4):
If partial monitoring allows to cut the monitoring overhead to a practical level in most cases, it also makes it likely, when using conventional partial monitoring techniques, to introduce inconsistencies in the reported event stream. To understand how such inconsistencies may be introduced, one needs to gain a basic understanding what we mean by an object-oriented system in the context of programming languages. A class or type is the building block of an object-oriented language and is a template that describes the data and behavior associated with instances of that class. When one instantiates a class, an object that is created looks and feels like other instances of the same class. The data associated with a class or an object is stored in member variables. The behavior associated with a class or object is implemented with methods. For example, a rectangle can be considered a class that has two attributes or member variables: length and breadth, and a method named area. From this rectangle class, one can create or instantiate new instances or new rectangle

objects. These new objects share similar behavior with respect to the attributes and methods that can be invoked on them. However, the actual values of the attributes and the value returned by the methods can be different. Method invocations are executed in the context of a thread of execution.

Detailed Description Text (5):
During the execution of an object-oriented program, potentially many objects are created, manipulated, and returned to system storage when they are no longer needed. This return to system storage can be automatic or through an explicit return-to-storage operation by the program. We refer herein to both types of return as reclaiming or reclamation. An object-oriented program achieves its task by potentially creating many objects of different classes and invoking methods on them.

Detailed Description Text (6):
Consider a visual representation of an object-oriented program execution through time as shown in FIG. 2. For the sake of discussion, let us assume that the identity of objects is nothing but the memory address to which they have been allocated. During its execution, the program creates 5 objects, OBJ1 through OBJ5, and invokes methods on these objects. However, OBJ1 and OBJ5 have the same identity since OBJ5 is allocated at the same address as OBJ1 after OBJ1 has been reclaimed. Since reporting was turned off during the time the OBJ1 was reclaimed and OBJ5 was created, a program understanding tool can read the monitoring events generated by a conventional monitoring system and mistakenly associate the costs and method invocations to the wrong objects. The consequence is that tools have to shield themselves from these inconsistencies, either by ignoring them and presenting potentially erroneous results or by implementing costly detection and prevention mechanisms.

Detailed Description Text (7):
The present invention provides an automated methodology for avoiding inconsistencies, while not limiting filtering mechanisms of partial monitoring. The invention is applicable to any system implementing some form of monitoring. The invention will be explained in the context of a monitoring API (application programming interface) relying on events to report internal activity to external listener subsystems. However, the invention is not so limited. That is, the invention may also, for example, apply to the internal design of systems that embed one or more kinds of listener subsystems.

Detailed Description Text (11):
It is also important to notice that, because we assume a programmatic monitoring API, entity events have to introduce an identity for the created entities, called the monitoring identity. The rationale for the identity is to allow other events to be able to refer to entities. For instance, an event reporting a method invocation needs to refer to the receiver object, the class implementing the method, and possibly even the thread on which the invocation occurs. Without monitoring identity, this would be impossible.

Detailed Description Text (13):
To further explain what we mean by object-oriented monitoring through events, let us define in abstract terms the concepts and events of a very typical object-oriented system. This definition applies to substantially any existing object-oriented system. However, it is to be appreciated that the invention is not intended to be limited by this abstract definition. An object-oriented system presents four basic entities: (i) type; (ii) object; (iii) thread; and (iv) invocation. A type describes an object structure and specifies its behavior, i.e., the set of methods one may invoke on an object of that type. An invocation is the execution of a method on its receiver object. Each invocation is carried on one and only one thread. Invocations nest, forming the thread execution stack.

Detailed Description Text (14):
Typically, such a system would have entity events to report the creation and reclamation of these entities. For instance, events reporting the creation and reclamation of an object or a thread, events for reporting the loading and unloading of a type, and/or events for reporting the beginning or end of an invocation. These

events may represent the core monitoring. That core would then typically be extended with extra activity events such as an event reporting that a thread is suspended or resumed. Other events may be used to report object management activities such as compaction, probably yielding new monitoring identities for objects. Synchronization events, reporting the "enters" (entries) and "leaves" (departures) from a critical section, as well as potential waits, would also be examples of activity events. A critical section is a set of program instructions that need to be executed as a unit with respect to the data structures they manipulate. In a system having a single "thread" of execution, achieving a critical section is trivial. But in modern systems, many threads of execution are concurrently executed to improve functionality and processing time. It is quite possible these threads need to access and manipulate shared data structures. To maintain consistency, a lock structure is associated with the shared data and a thread performs the operations in a critical section only after gaining ownership to the lock. Once the operations are completed, the thread releases the ownership of the lock. If it happens that another thread has lock ownership, then a thread desiring to acquire the lock waits on the owning thread until the lock is free. A diagram showing the different threads in a system with arrows drawn from a thread waiting for a resource to the thread owning the resource is called a wait graph. Such a wait graph, with accurate information and some additional summary information, is an excellent tool for determining sources of contention and reasons for infinite waiting of threads for resources.

Detailed Description Text (17):
Each of these cases may provoke the erroneous reporting of potentially serious conditions from conventional monitoring tools that support some form of partial monitoring. Such erroneous reporting may include, for example, erroneous profiling information (accounting of times), erroneous object reclamation, erroneous memory leaks, or even false deadlocks. Unfortunately, there exist no comprehensive solutions for these and other problems in the prior art in the area of supporting partial and consistent monitoring for object-oriented systems. In general, the conventional program execution environments leave the burden of making sense of the information generation to the information processing tools. In the context of Java language (see, e.g., "Java Language Specification," J Gosling, B. Joy and G. Steele, Addison Wesley, ISBN 0201634511 (1996); and "Java 1.1 Developer's Handbook," P. Heller, S. Roberts, with P. Seymour and T. McGinn, Sybex, ISBN 0-7821-1919-0), the Java Virtual Machine (JVM) from Sun Microsystems exports an API called JVMPI (see, e.g.,"Comprehensive profiling support in the Java Virtual Machine," Sheng Liang and Deepa Viswanathan. Usenix Conference on Object-Oriented Technologies (COOTS) 1999) that allows for callbacks into the JVM to handle certain inconsistencies related to the type system. However, it is known to be incomplete. In particular, the trace generation can be incorrect in the presence of garbage collection. In the context of the C language, some work has been done in the context of a tool named Parasight (see, e.g.,"Non-intrusive and interactive profiling in Parasight," Ziya Aral and Ilya Gemter, Proceedings of the ACM/SIG PLAN PEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems, pages 21-30. July 1988) that provides some support for consistent tracing. However, they do not address the issues due to garbage collection since it does not exist for normal C programs.

Detailed Description Text (18):
In a significant departure over the prior art, the present invention provides methodologies for supporting partial and consistent monitoring of object-oriented programs. These methods, when incorporated in program execution environments, can be used to produce consistent information which can be used by batch and interactive tools such as, for example, program analyzers, visualizers and debuggers for purposes such as program understanding, visualization and debugging.

Detailed Description Text (19):
For the description of this embodiment of the invention, we will assume without loss of generality that a virtual machine (VM) executes the object-oriented system of interest and events occurring as part of the system execution are delivered to external event listeners which have to register to a single event source, exported by the running VM. Notice that the VM does not take care of maintaining any history of generated events. That would be the responsibility of listeners.

Detailed Description Text (20):

The invention provides a monitoring methodology that is composed of two parts: (i) a categorization of events that fully describe the execution of an object-oriented program; and (ii) a set of sequencing rules in order to ensure the consistency of the event stream. These parts will be explained in detail below. Referring back to FIG. 1, it is to be appreciated that this inventive methodology may be implemented by the monitoring agent 8 (also referred to herein as the monitoring system or subsystem) within the program execution environment 4. The object-oriented program would then be program 2 in FIG. 1. The event stream is reported by the monitoring agent 8 to, for example, the program event processing tool 10. Reporting may be controlled by the event reporting controller 12.

Detailed Description Text (24):
Type events are the definition of types allowing -to describe objects which are instances of types. A type is either a basic type, a class or an interface. Basic types are the classical ones such as integers, floats, etc. A class has a name, a class it is derived from (called super class), a list of implemented interfaces, a set of methods that it implements, and a set of fields it defines. An interface has a name, a set of interfaces it extends, and a set of methods and constants it declares. A method has a return type, a name, and a list of parameter types (the receiver being implicit and compatible with the class implementing the method). A field has a name and a type.

Detailed Description Text (25):
We define three events on types: create, load, and reclaim. The rationale for three events is further elaborated in the following section (Sequencing Rules). The create event specifies the name of the type and monitoring identity. The load event defines the type. It refers to the super class and to the implemented interfaces. It also includes the description of the class members, that is, the methods and the fields. Method descriptions refer to the return and parameter types. Field description refers to the field type. The reclaim event indicates the type is no longer known to the monitored system.

Detailed Description Text (26):
Object events report the life cycle of objects (creation, destruction) as well as the object graph, that is, references between objects. Objects are created upon explicit request and later reclaimed. Whenever an object is allocated, a create event is generated; whenever an object is reclaimed, a reclaim event is generated. The create event includes the monitoring identity of the created object and refers to the class of the object. The reclaim event just refers to the reclaimed object.

Detailed Description Text (27):
A reference identifies at runtime an object and allows an object A to refer to an object B. A reference event reports the existence of a reference from a pointing-to object to a pointed-to object (potentially identical). A reference event refers to the field (and may also refer to its declaring class) containing the reference and to the referred-to object.

Detailed Description Text (28):
Execution events report the procedural aspect of the monitored system, although retaining its object-oriented characteristics. First of all, two events report the creation and destruction of a thread. The create event provides the thread name and its monitoring identity. The destruction event just refers to the reclaimed thread. A renaming event is also provided allowing to keep track of name changes.

Detailed Description Text (29):
A thread executes method invocations following a Last-In-First-Out (LIFO) model. In other words, if a thread executes a method A that invokes another method B, then the LIFO model states the execution of method B completes before the execution of method A. It can be said that method A is the parent of method B. A method invocation represents the execution of the method instructions. Two events are used to report method invocation: an enter and a leave event. The enter event refers to the thread, the class implementing the currently executing method, and the receiver object. The leave event only refers to the thread. This is sufficient because of the LIFO model.

Detailed Description Text (34):
Although these two rules seem very intuitive, their enforcement is tricky in some cases. We explained earlier that we define three events for types: the create, load, and reclaim events. The rationale is a sequencing one. Type definitions are by definition recursive and potentially cyclic. This suggests to separate the definition of a type from its mere existence so to be able to break cyclic dependencies in type definitions. The create event must precede the load event which must precede the reclaim event. Any other event referring to a class must appear after the load event for that class. For certain languages, maintaining such a precedence invariant in the VM can be tricky. For instance, consider the static initializer feature in the Java language. Static initializers are snippets of code which initialize class static fields at load time. Care must be taken to issue the class load event before any static initializer is run, otherwise the sequencing rules would be violated, having events such as enter and leave events referring to a non-loaded class.

Detailed Description Text (38):
The third consistency rule (3. above) is not surprising and quite obviously complements the normal sequencing rules. It states that if an entity has been defined, then its reclamation must be reported to. The rationale is to allow listeners to maintain accurate mapping between monitoring identities and entities. See an example of such inconsistencies below. Full reporting status is assumed at the beginning of the example. Step (1) Class C1 is created with id #12 Step (2) Class C1 is loaded, has methods M1 (id #23 and M2 #24) Step (3) Object O1 (id #34) of Class C1 is created Step (4) Invocation of method M1 on O1 Step (5) The user turns monitoring off Step (6) Object O1 is reclaimed (id #34 is freed for reuse) Step (7) Object O2 (id #34 is reused) of class C1 Step (8) The user turns tracing back on . . . Step (9) Invocation of method M1 on O2 is executed

Detailed Description Text (46):
Bookkeeping: This generic name is used to cover operations that have a notion of summary associated with them. For example, a counter for the number of class-creation events may need to be incremented even if the specific class object in the class-creation event is not in the current selection.

Detailed Description Text (49):
Reported Entity Creation Set (RECS): This is the subset of all the entities that are currently live and have been reported by the monitoring system to external listener subsystems. For instance, if the creation of a class C has been reported, then it becomes a member of this set. If the class C is reclaimed it is removed from this set.

Detailed Description Text (51):
Identity Event Set (IES): This is a special kind of an event set that is defined to handle certain technical problems in reporting events. In certain languages, it is possible to have cycles in the type system. In other words, to completely define a type A, one may eventually need to define a type B that in turn requires the definition of type A. In such a scenario, it is not possible to order the reporting of the definitions of type A and B. To break this cycle, we introduce the notion of an "identity event" for a type which can contain information about its identity and all other information excluding those parts that require knowledge of other types. For the above example, the identity event can contain identity of class A, the name of the class, and may be the location of the file from which the class definition was loaded into the execution environment. For a type A, this is the set of identity events that need to be reported before the type can be defined. An Unreported Identity Event Set (UIES) for an event is the subset of IES information that has not yet been reported.

Detailed Description Text (62):
Referring now to FIG. 6, a block diagram illustrating an exemplary computer system for implementing this invention is shown. The computer system may comprise a processor 602 operatively coupled to memory 604 and I/O devices 606. It is to be appreciated that the term "processor" as used herein is intended to include any processing device, such as, for example, one that includes a CPU (central processing unit). The term "memory" as used herein is intended to include memory associated

with a processor or CPU, such as, for example, RAM, ROM, a fixed memory device (e.g., hard drive), a removable memory device (e.g., diskette), flash memory, etc. In addition, the term "input/output devices" or "I/O devices" as used herein is intended to include, for example, one or more input devices, e.g., keyboard, for inputting data to the processing unit, and/or one or more output devices, e.g., CRT display and/or printer, for presenting results associated with the processing unit. It is also to be understood that "processor" may refer to more than one processing device and that various elements associated with a processing device may be shared by other processing devices. Accordingly, software components including instructions or code for performing the methodologies of the invention, as described herein, may be stored in one or more of the associated memory devices (e.g., ROM, fixed or removable memory) and, when ready to be utilized, loaded in part or in whole (e.g., into RAM) and executed by a CPU. Thus, in accordance with this exemplary implementation, it is to be understood that one or more of the elements shown in FIG. 1 may be implemented on a computer system as illustrated in FIG. 6.

Detailed Description Text (63):
While we have explained the methodologies of the invention generally for application to a large class of problems, it is to be appreciated that one need not implement the invention in the same manner as described in the illustrative embodiments. For example, instead of explicitly creating the many sets described, one can extend the object representation supported by a program execution environment to carry additional flags to denote set membership, thus reducing the complexity and performance overheads of set management to simple setting and resetting of flags in objects. Some program execution environments are such that once objects are allocated in memory, their raw memory address remains the same all through the program execution. In such cases, the need for generating new identities for entities is obviated, as well as the memory for storing identities. Further, instead of building a complete RET before generating events, one can do a depth first expansion and reporting of the RET which can reduce space overheads. Further, if the program execution system invokes the monitoring subsystem along different paths for different events, then the context -in which a set of operations is executed is known statically. This feature can be used to eliminate run-time checks for the type of the event generated by the program execution system.

Detailed Description Text (64):
Accordingly, it is to be understood that the methodologies of the invention can be readily applied to the generation of a consistent set of monitoring events in the presence of partial monitoring. Further, the consistency is maintained even when the information subsetting criteria are dynamically changed during program execution. The methodologies are general enough to apply to a large class of systems that can be depicted, in an object-oriented manner, thus not necessarily to just object-oriented systems. We have applied this invention for supporting consistent partial monitoring for programs written in the Java programming language by modifying a version of Java Virtual Machine (JVM). Further, we applied many of the optimizations suggested above.

Other Reference Publication (3):
S. Liang et al., "Comprehensive Profiling Support in the JAVA.TM. Virtual Machine," USENIX Association, 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99), pp. 229-240.

CLAIMS:

1. A method of monitoring events generated by an object-oriented program, the method comprising the steps of: monitoring events which describe executed operations associated with the object-oriented program; and applying one or more sequencing rules when reporting a subset of the monitored events, the one or more sequencing rules substantially ensuring consistent reporting of the subset of monitored events; wherein when the monitoring step appears to be partially or totally disabled from an external perspective, based on previously monitored events, the step of applying one or more sequencing rules may cause one or more events to be reported to preserve consistency.

16. The method of claim 1, wherein the monitoring step includes monitoring events

generated in association with the object-oriented program, as the program is executed by an object-oriented system.

19. Apparatus for monitoring events generated by an object-oriented program, the apparatus comprising: at least one processor operative to: (i) monitor events which describe executed operations associated with the object-oriented program; and (ii) apply one or more sequencing rules when reporting a subset of the monitored events, the one or more sequencing rules substantially ensuring consistent reporting of the subset of monitored events; wherein when the monitoring operation appears to be partially or totally disabled from an external perspective, based on previously monitored events, the operation of applying one or more sequencing rules may cause one or more events to be reported to preserve consistency.

25. An article of manufacture for monitoring events generated by an object-oriented program, comprising a machine readable medium containing one or more programs which when executed implement the steps of: monitoring events which describe executed operations associated with the object-oriented program; and applying one or more sequencing rules when reporting a subset of the monitored events, the one or more sequencing rules substantially ensuring consistent reporting of the subset of monitored events; wherein when the monitoring step appears to be partially or totally disabled from an external perspective, based on previously monitored events, the step of applying one or more sequencing rules may cause one or more events to be reported to preserve consistency.

26. A computer program product for monitoring events generated by an object-oriented program, the computer program product comprising: first instruction means for monitoring events which describe executed operations associated with the object-oriented program; and second instruction means for applying one or more sequencing rules when reporting a subset of the monitored events, the one or more sequencing rules substantially ensuring consistent reporting of the subset of monitored events; wherein when the first instruction means for monitoring appears to be partially or totally disabled from an external perspective, based on previously monitored events, the second instruction means for applying one or more sequencing rules may cause one or more events to be reported to preserve consistency.

WEST

Generate Collection Print

L8: Entry 3 of 8 File: USPT Feb 18, 2003

DOCUMENT-IDENTIFIER: US 6523168 B1
TITLE: Reduction of object creation during string concatenation and like operations that utilize temporary data storage

Brief Summary Text (11):
Compilation of a string concatenation statement in a Java source code program by a Java compiler results in the generation of program code that utilizes a temporary mutable string object, known in Java as a "StringBuffer" object, in performing the string concatenation operation. As an example, Table I below illustrates an exemplary "myExample" Java class that includes an "exampleConcat" procedure that receives two arguments "s1 " and "s2" and returns a "result" object that is the concatenation of the data in the "s1" and "s2" arguments:

Brief Summary Text (15):
String concatenation operations are used extensively in a number of Java applications such as manipulating results from database files, and generating dynamic documents (e.g., hypertext markup language (HTML) documents and the like), among others. As such, it is possible in many applications for string concatenations to result in the creation of a relatively large number of temporary objects, which can have a significant negative impact on overall system performance.

Brief Summary Text (19):
In one specific, but by no means exclusive, implementation, a reusable temporary object is provided in the form of a mutable string object, which is utilized in the performance of string concatenation operations in the Java programming environment. Consequently, rather than creating a new mutable string object (as well as an underlying character array object) for each string concatenation operation, an existing mutable string object, allocated at the initialization of a program (or a thread thereof), is used as the temporary storage for each operation. The total number of objects created as a result of multiple string concatenation operations is therefore reduced, easing allocation and collection overhead, and accordingly improving overall system performance.

Brief Summary Paragraph Table (1):
TABLE I Example String Concatenation public class myExample { public String exampleConcat(String s1, String s2) { String result = s1 + s2; return (result); } }

Detailed Description Text (3):
A reusable temporary object consistent with the invention may be used in the performance of a wide variety of computer operations that require some form of temporary data storage. For example, the specific implementation discussed hereinafter focuses on the utilization of a reusable temporary object in connection with performing string concatenation operations, e.g., in a Java or other object-oriented environment. However, it will be appreciated that other string-based operations may also utilize a reusable temporary object consistent with the invention. Furthermore, the invention may also have benefit when used with other types of computer operations that rely on temporary data storage.

Detailed Description Text (6):
A translation program consistent with the invention may incorporate a compiler to generate suitable program code for reusing a reusable temporary object during compilation of source code into either an executable or an intermediate representation. For example, in the specific implementation described hereinafter,

the generation of program code is performed by a Java compatible compiler when generating a Java bytecode or intermediate representation, i.e., a class file, from a Java source code program. However, it will also be appreciated that program code generation may also be performed when compiling a computer program into native executable program code, e.g., in the c++ programming language, or when compiling Java bytecodes into native code with a static compiler, among other environments. Furthermore, program code generation consistent with the invention may also be performed during interpretation of a computer program, e.g., when interpreting Java bytecodes into corresponding native instructions, and thus a translation program consistent with the invention may include an interpreter (e.g., a Java virtual machine) in lieu of or in addition to a compiler. Moreover, in some environments compilation may be performed "just-in-time", and as such a just-in-time compiler may also be used to implement all or a portion of the functionality of a translation program consistent with the invention. Furthermore, given the generation of program code may be performed at one or more of the above-described stages, it will be appreciated that the computer program being translated, as well as the computer program being generated as a result of the translation process, may be utilized using any number of representations, whether human or machine readable in nature. Consequently, the invention should not be limited to the compilation of human readable source code into an intermediate class file representation as is specifically described hereinafter.

Detailed Description Text (14):
Computer 30 operates under the control of an operating system 40, and executes or otherwise relies upon various computer software applications, components, programs, objects, modules, data structures, etc. (e.g., compiler 42, virtual machine 44, source code 46 and class files 48, among others). Moreover, various applications, components, programs, objects, modules, etc. may also execute on one or more processors in another computer coupled to computer 30 via a network 38, e.g., in a distributed or client-server computing environment, whereby the processing required to implement the functions of a computer program may be allocated to multiple computers over a network.

Detailed Description Text (15):
In general, the routines executed to implement the embodiments of the invention, whether implemented as part of an operating system or a specific application, component, program, object, module or sequence of instructions will be referred to herein as "computer programs", or simply "programs". The computer programs typically comprise one or more instructions that are resident at various times in various memory and storage devices in a computer, and that, when read and executed by one or more processors in a computer, cause that computer to perform the steps necessary to execute steps or elements embodying the various aspects of the invention. Moreover, while the invention has and hereinafter will be described in the context of fully functioning computers and computer systems, those skilled in the art will appreciate that the various embodiments of the invention are capable of being distributed as a program product in a variety of forms, and that the invention applies equally regardless of the particular type of signal bearing media used to actually carry out the distribution. Examples of signal bearing media include but are not limited to recordable type media such as volatile and non-volatile memory devices, floppy and other removable disks, hard disk drives, magnetic tapes, optical disks (e.g., CD-ROM's, DVD's, etc.), among others, and transmission type media such as digital and analog communication links.

Detailed Description Text (16):
In addition, various programs described hereinafter may be identified based upon the application for which they are implemented in a specific embodiment of the invention. However, it should be appreciated that any particular program nomenclature that follows is used merely for convenience, and thus the invention should not be limited to use solely in any specific application identified and/or implied by such nomenclature.

Detailed Description Text (19):
The specific embodiment described hereinafter focus on a particular application of the invention in optimizing the performance of computer programs executed in the Java programming environment developed by Sun Microsystems. However, it should be

appreciated that the invention may have applicability in other programming environments that utilize temporary objects in performing computer operations, particularly string operations such as string concatenations and the like.

Detailed Description Text (20):
FIG. 3 illustrates the primary software components utilized in the illustrated embodiment. Specifically, FIG. 3 shows a compiler 42 coupled to a virtual machine 44, with the compiler receiving as input source code 46 and outputting in response thereto one or more class files 48 capable of being executed by virtual machine 44.

Detailed Description Text (22):
The generated bytecodes are organized into one or more classes, representing the templates for the objects that are to be allocated and utilized during execution of the computer program by virtual machine 44. The classes are organized into class files 48 containing both the executable bytecodes and data relied upon by such executable code. Other information about an object is also typically included within the class file, as is known in the art.

Detailed Description Text (23):
Once generated, class files may be distributed to third parties and/or stored to some persistent medium for later execution, or may be immediately executed by virtual machine 44. Virtual machine 44 implements a Java Virtual Machine (JVM), which essentially emulates the operation of a hypothetical microprocessor on a specific computer platform. Different virtual machines may be utilized to permit the class files to be executed on different platforms. Moreover, it should be appreciated that compiler 42 and virtual machine 44 need not reside on the same computer system.

Detailed Description Text (24):
Class files are loaded by virtual machine 44 using a class loader component 54, in a manner generally known in the art. Once loaded, the classes in the class files are processed by a class verification block 52, which performs various verification and analysis operations on the classes to ensure that the classes are error free and will not cause various run-time errors, as well as that all security requirements of the Java language are met.

Detailed Description Text (25):
To provide working storage during execution of the computer program, virtual machine 44 includes an object heap 56. Data storage is allocated during runtime, and is periodically cleaned up (with unused objects discarded) by garbage collection logic represented at 62. Any number of known garbage collection schemes may be used consistent with the invention.

Detailed Description Text (26):
In response to the loading and verification of classes in blocks 54 and 52, one or more threads 60 are executed by an interpreter 58 that generates for each bytecode suitable native code appropriate for the platform upon which the virtual machine executes. It will be appreciated that the interpretation and execution of Java bytecodes by virtual machine 44 are operations that are well known in the art. Additional modifications to the virtual machine, including just-in-time compilation, among other alternatives, may also be implemented in virtual machine 44 consistent with the invention.

Detailed Description Text (27):
In the illustrated implementation of FIG. 3, the optimization described herein is implemented within bytecode generation block 50 of compiler 42, which has the benefit of requiring no specific modifications to the virtual machine or to any Java computer programs or class files. In other implementations, however, the optimization may be performed in other stages of compiler 42, as well as during interpretation by virtual machine 44 or during compilation by a static compiler that compiles the Java class files generated by compiler 42 into native instructions for a particular computer platform. The optimization relies on one or more temporary StringBuffer objects 64, illustrated as resident in object heap 56.

Detailed Description Text (28):

FIG. 4 illustrates the general program flow of bytecode generation block 50 in greater detail. After routine initialization in block 68 (which may include, for example, retrieval of the source code from mass storage), a loop is initiated in block 70 to process each statement in the source code program provided to the compiler. Block 70 retrieves the next unprocessed statement from the source code program. Next, block 72 determines whether the statement is to create a new execution thread. If not, control passes to block 74 to determine whether the statement is a string concatenation statement. If not, control passes to block 76 to handle the statement in a conventional manner, as is well known in the art. Control then passes to block 78 to determine whether additional unprocessed statements exist in the source code, and if so, control returns to block 70 to process the next statement. Once all statements have been processed, block 78 terminates the bytecode generation process.

Detailed Description Text (29):
Returning to block 72, in the illustrated implementation, one reusable temporary StringBuffer object is utilized for each execution thread of a Java computer program. As such, whenever a statement to initialize a new thread is detected in the source code, block 72 passes control to block 80 to generate temporary StringBuffer initialization code for the resulting class file to initialize a new temporary StringBuffer object (also identified hereinafter as a "tsb" object) for the thread. One suitable source code representation of temporary StringBuffer initialization code is shown below in Table III:

Detailed Description Text (30):
The initialization code can initially allocate any size of reusable temporary StringBuffer object, since the StringBuffer class will automatically allocate additional memory for a StringBuffer object as it is needed. However, to minimize the allocation of additional memory during the performance of string concatenation operations, it may be desirable in some implementations to initially allocate a relatively large StringBuffer object. It will be appreciated that the generation of bytecodes corresponding to the above source code representation is well within the ability of one of ordinary skill in the art having the benefit of the instant disclosure.

Detailed Description Text (31):
Returning to block 80 of FIG. 4, once the temporary StringBuffer initialization code is generated, additional program code is generated in block 82 for initializing the thread, in a manner well known in the art. Control then returns to block 78.

Detailed Description Text (39):
Of the operations represented by the statements in Table IV, only the "toString( )" necessarily creates an object, compared to the minimum of three objects that are necessarily created by conventional string concatenation usage code (as described above in connection with Table II). Thus a reduction in object allocations and deallocations associated with string concatenation operations of up to 66% may be realized in many applications. Particularly in applications where numerous string concatenation operations are used, the performance benefits can be substantial.

Detailed Description Text (42):
With the program code of Table V, an additional boolean flag "TsbInUse" is added to the Thread class (typically by extending the Thread class, as represented by the class "ExtendedThread"), indicating whether the reusable temporary StringBuffer object is currently in use. If so, conventional string concatenation code is executed. If not, program code corresponding to Table IV is executed, with a lock implemented by first setting, and then resetting the flag once the string concatenation operation is complete (e.g., using a "setTsbInUse( )" method provided in the extended Thread class "ExtendedThread"). Other known manners of implementing a lock or other synchronization mechanism may also be used in the alternative. Furthermore, rather than extending the Thread class, other manners of providing an in use flag that is globally-available to a thread may also be used. In addition, rather than determining the current thread during each string concatenation operation (in line 2 of Table V), a current thread identifier could be obtained during thread initialization.

Detailed Description Text (43):
It should also be appreciated that in single threaded applications, only one reusable temporary StringBuffer object may be required. Furthermore, allocation of the object may be performed at different instances, e.g., at the beginning of the main( ) routine for a compiled Java program. Even in multi-threaded applications, a single object could be used, with some form of synchronization mechanism (e.g., a lock) used to synchronize access to the reusable temporary object.

Detailed Description Text (44):
Furthermore, in other implementations, it may be desirable to utilize a pool of reusable temporary StringBuffer objects for use either by individual threads or by all threads executing in a program. For example, Tables VI and VII below illustrate additional alternate temporary StringBuffer initialization and usage code implementations suitable for selecting between multiple reusable temporary StringBuffer objects, wherein a stack object referred to as "tsbStack" is used to store the pool of StringBuffer objects. Moreover, it is assumed that each thread has its own stack, and thus that no synchronization between threads is required for each stack of StringBuffer objects.

Detailed Description Paragraph Table (1):
TABLE III Temporary StringBuffer Initialization Code StringBuffer tsb = new StringBuffer();

Detailed Description Paragraph Table (4):
TABLE VI Alternate Temporary StringBuffer Initialization Code Stack tsbStack = new Stack();

CLAIMS:

13. The method of claim 1, further comprising, for each of the plurality of operations defined in the first computer program that require the use of temporary storage, generating program code in the second computer program to initialize the reusable temporary object prior to using the reusable temporary object in performing the operation.

16. A method of compiling Java source code into at least one Java class file, the method comprising: (a) generating program code for the Java class file that allocates a reusable temporary stringbuffer object; and (b) for each of a plurality of string concatenation statements in the Java source code, each of which including a plurality of arguments, generating program code in the Java class file that initializes the reusable temporary stringbuffer object and appends each argument in the string concatenation statement to the reusable temporary stringbuffer object.

29. The apparatus of claim 17, wherein the translation program is further configured to generate, for each of the plurality of operations defined in the first computer program that require the use of temporary storage, program code in the second computer program to initialize the reusable temporary object prior to using the reusable temporary object in performing the operation.